In [1]:
```
# Import Libraries
import pandas as pd
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers
from sklearn.model_selection import train_test_split
import seaborn as sns
from matplotlib import pyplot as plt

# Load the TensorBoard notebook extension.
%load_ext tensorboard
from datetime import datetime
from packaging import version

print("TensorFlow version: ", tf.__version__)
assert version.parse(tf.__version__).release[0] >= 2, \
    "This notebook requires TensorFlow 2.0 or above."
import tensorboard
tensorboard.__version__
```

```
2023-02-09 11:10:01.236746: I tensorflow/core/platform/cpu_feature_guard.c
c:193] This TensorFlow binary is optimized with oneAPI Deep Neural Network
Library (oneDNN) to use the following CPU instructions in performance-criti
cal operations:  SSE4.1 SSE4.2 AVX AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate
compiler flags.
TensorFlow version:  2.10.0
```

Out[1]:
```
'2.10.0'
```

Import and Understand the Data

I am working with the vehicle emissions data found at https://sumo.dlr.de/docs/Simulation/Output/EmissionOutput.html

The imported column values are: time_step, ID, eclass, CO2, CO, HC, NOx, PMx, fuel, electricity, noise - in dB, route - name of the route, type - name of the vehicle, waiting, lane - the name of the lane where the vehicle is moving, pos - position, speed, angle - the vehicles angle, pos_x - the absolute x coordinate, pos_y - the absolute y coordinate

In [2]:
```
emission_train = pd.read_csv("UC-Emission.csv", delimiter=",", quoting = 3)
```

In [3]:
```
# Next, I want to see an example of the data I am working with to ensure it
display(emission_train.head(100))

# I also want to view some descriptive statistics to look for the possibilit
# data lies, and further explanation of the data.
display(emission_train.describe())
```

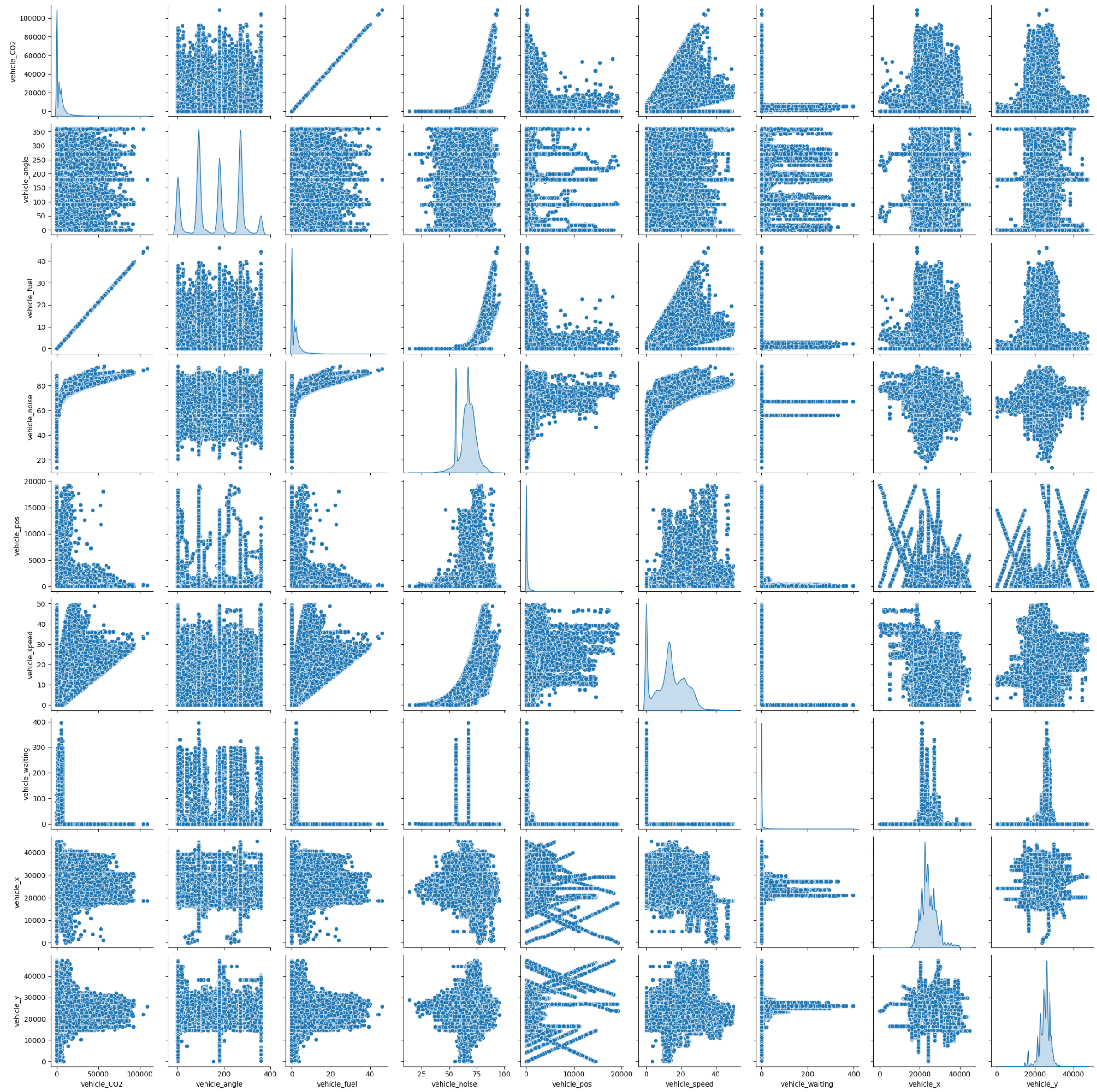
	timestep_time	vehicle_CO	vehicle_CO2	vehicle_HC	vehicle_NOx	vehicle_PMx	vehicle_angl
0	0.0	15.20	7380.56	0.00	84.89	2.21	50.2
1	0.0	0.00	2416.04	0.01	0.72	0.01	42.2
2	1.0	17.92	9898.93	0.00	103.38	2.49	50.2
3	1.0	0.00	0.00	0.00	0.00	0.00	42.2
4	1.0	164.78	2624.72	0.81	1.20	0.07	357.0
...	...	...	...	...	...	...	.
95	7.0	23.44	2578.06	0.15	0.64	0.05	0.1
96	7.0	732.32	18759.70	3.34	3.79	1.19	179.9
97	7.0	294.68	6949.38	1.29	1.47	0.43	179.9
98	7.0	236.07	4292.19	0.97	0.93	0.30	1.9
99	7.0	179.19	1228.61	0.64	0.31	0.17	180.0

100 rows × 20 columns

	timestep_time	vehicle_CO	vehicle_CO2	vehicle_HC	vehicle_NOx	vehicle_PMx	ve
count	1.633101e+07	1.633101e+07	1.633101e+07	1.633101e+07	1.633101e+07	1.633101e+07	1.
mean	4.112561e+03	5.764304e+01	4.919050e+03	7.284125e-01	1.769589e+01	4.227491e-01	1.
std	2.168986e+03	8.854365e+01	7.959043e+03	1.589816e+00	5.993168e+01	1.164065e+00	1.
min	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.
25%	2.291000e+03	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	9.
50%	4.133000e+03	2.017000e+01	2.624720e+03	1.500000e-01	1.200000e+00	6.000000e-02	1.
75%	5.903000e+03	1.034400e+02	6.161010e+03	7.600000e-01	2.710000e+00	1.500000e-01	2.
max	1.441800e+04	3.932950e+03	1.153026e+05	1.729000e+01	8.864200e+02	1.432000e+01	3.

In [4]:
```
# It can also be helpful to view correlation datasets visually to look for c
# correlations to each other. We only want 5% of the data, as the correlatic
# taxing. Since there are over 16 million shuffled rows, 5% will be adiquate

correlation_graph_data = emission_train.sample(frac=0.05).reset_index(drop=T
print(len(emission_train), 'emission_train')
print(len(correlation_graph_data), 'correlation_graph_data')

sns.pairplot(correlation_graph_data[['vehicle_CO2', 'vehicle_angle', 'vehicl
del correlation_graph_data

# In this case we can see that several features have linear relationships, s
# others appear not to be strongly related.
```

```
16331008 emission_train
816550 correlation_graph_data
```


vehicle_CO2
vehicle_angle
vehicle_fuel
vehicle_noise
vehicle_pos
vehicle_speed
vehicle_waiting
vehicle_x
vehicle_y

Clean up the Dataset

This dataset is already mostly clean, but for the purposes of machine learning, it has many features that will not be helpful to us, and contains NaN values. It is at this point that we will want to remove data that will not be useful for the purpose of prediction. This can be done in many ways but first of all, since we are only predicting vehicle CO2, we can remove the other emissions. The case could also be made that the other emission classes could have relationships with CO2 production, but that relationship would be confounding as it can be expected that, due to the nature of combustion, that with CO2 production we will also have CO, HC, NOx, and PMx. The computer may learn to just estimate the other emissions production which we do not want.

Further, we can also get rid of the timestep_time, vehicle_id, vehicle_angle, vehicle_lane, vehicle_pos, vehicle_route, vehicle_x, and vehicle_y. These are removed as they do not effect the CO2 production and increase training time and complexity. This decision was concluded after viewing the correlational graphs as well as experimenting with feature input during the training process. Vehicle electricity was also removed as it reduced the models overall accuracy in both the validation and test sets.

In [5]:
```
emission_train = emission_train.drop(columns=["vehicle_CO", "vehicle_HC", "v
                                               "timestep_time", "vehicle_id",
                                              "vehicle_route", "vehicle_x", '

for header in ["vehicle_eclass", "vehicle_type"]:
    emission_train[header] = emission_train[header].astype(str)

# Remove NaNs
emission_train = emission_train.dropna().reset_index(drop=True)

# Shuffle the dataset
emission_train_shuffle = emission_train.sample(frac=1)
```

Split Data for Machine Learning

Next, the data is split into training, validation and test sets for machine learning.

In [6]:
```
train_df, test_df = train_test_split(emission_train_shuffle, test_size=0.1)
train_df, val_df = train_test_split(train_df, test_size=0.2)

print(len(train_df), 'train examples')
print(len(val_df), 'validation examples')
print(len(test_df), 'test examples')

del emission_train
```

```
11758324 train examples
2939582 validation examples
1633101 test examples
```

Organize Features

Next, I have organized the input features into either numerical or categorical. This tells tensorflow which type of data it will be working with during training. Our feature layer can then be created.

In [7]:
```
feature_cols = []

# Numeric Columns
for header in ["vehicle_fuel", "vehicle_speed", "vehicle_waiting", "vehicle_
    num_cols = tf.feature_column.numeric_column(key = header)
    feature_cols.append(num_cols)

# Indicator Columns
indicator_col_names = ["vehicle_eclass", "vehicle_type"]
for col_name in indicator_col_names:
    categorical_column = tf.feature_column.categorical_column_with_vocabular
                                        train_df[col_name].unique())
    indicator_column = tf.feature_column.indicator_column(categorical_column
    feature_cols.append(indicator_column)

# Create a feature layer for tf
feature_layer = tf.keras.layers.DenseFeatures(feature_cols, name='Features')
```

Create and Train the Model

For this project I am using a general ANN model which utilizes the optimizer Adam, and the mean squared error metric for the loss function. As a final step I implemented the softplus activation layer which implements the softmax function.

After testing, I have found a 4-layer system (24, 16, 11, 5 units respectively), with a batch size of , and a learning rate of 0.00024, resulted in model convergence and reliable validation and test data prediction.

In [8]:

```
# Hyperparameters
learning_rate = 0.00024
epochs = 90
batch_size = 25000

# Label
label_name = "vehicle_CO2"
shuffle = True

#---sequential model---#
model = tf.keras.models.Sequential([
    feature_layer,
    # Hidden Layers
    tf.keras.layers.Dense(units=24,
                          activation='relu',
                          kernel_regularizer=tf.keras.regularizers.l1(l=0.03
                          name='Hidden1'),
    tf.keras.layers.Dense(units=16,
                          activation='relu',
                          kernel_regularizer=tf.keras.regularizers.l1(l=0.03
                          name='Hidden2'),
    tf.keras.layers.Dense(units=11,
                          activation='relu',
                          kernel_regularizer=tf.keras.regularizers.l1(l=0.03
                          name='Hidden3'),
    tf.keras.layers.Dense(units=5,
                          activation='relu',
                          kernel_regularizer=tf.keras.regularizers.l1(l=0.02
                          name='Hidden4'),
    # Output layer
    tf.keras.layers.Dense(units=1,
                          activation='softplus',
                          name='Output')

])

model.compile(optimizer=tf.keras.optimizers.Adam(lr=learning_rate),
              loss=tf.keras.losses.MeanSquaredError(),
              metrics=['mse'])

#---Training the Model---#
# Keras TensorBoard callback.
logdir = "logs/fit/" + datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=logdir)

train_lbl = np.array(train_df["vehicle_CO2"])
train_df = train_df.drop(columns=["vehicle_CO2"])
# Split the datasets into features and label.
train_ft = {name:np.array(value) for name, value in train_df.items()}

val_lbl =  np.array(val_df["vehicle_CO2"])
val_df = val_df.drop(columns=["vehicle_CO2"])
val_ft = {name:np.array(value) for name, value in val_df.items()}

# Keras TensorBoard callback.
logdir = "logs/fit/" + datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=logdir)
```

```
# Training function
model.fit(x=train_ft, y=train_lbl, batch_size=batch_size,
          epochs=epochs, verbose=1, validation_data=(val_ft, val_lbl), shuf
```

```
2023-02-09 11:13:10.428599: I tensorflow/core/platform/cpu_feature_guard.c
c:193] This TensorFlow binary is optimized with oneAPI Deep Neural Network
Library (oneDNN) to use the following CPU instructions in performance-criti
cal operations:  SSE4.1 SSE4.2 AVX AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate
compiler flags.
2023-02-09 11:13:10.429143: I tensorflow/core/common_runtime/process_util.c
c:146] Creating new thread pool with default inter op setting: 2. Tune usin
g inter_op_parallelism_threads for best performance.
/home/michael/anaconda3/lib/python3.9/site-packages/keras/optimizers/optimi
zer_v2/adam.py:114: UserWarning: The `lr` argument is deprecated, use `lear
ning_rate` instead.
  super().__init__(name, **kwargs)
```

```
Epoch 1/90
WARNING:tensorflow:Layers in a Sequential model should only have a single i
nput tensor. Received: inputs={'vehicle_eclass': <tf.Tensor 'IteratorGetNex
t:0' shape=(None,) dtype=string>, 'vehicle_fuel': <tf.Tensor 'IteratorGetNe
xt:1' shape=(None,) dtype=float32>, 'vehicle_noise': <tf.Tensor 'IteratorGe
tNext:2' shape=(None,) dtype=float32>, 'vehicle_speed': <tf.Tensor 'Iterato
rGetNext:3' shape=(None,) dtype=float32>, 'vehicle_type': <tf.Tensor 'Itera
torGetNext:4' shape=(None,) dtype=string>, 'vehicle_waiting': <tf.Tensor 'I
teratorGetNext:5' shape=(None,) dtype=float32>}. Consider rewriting this mo
del with the Functional API.
WARNING:tensorflow:Layers in a Sequential model should only have a single i
nput tensor. Received: inputs={'vehicle_eclass': <tf.Tensor 'IteratorGetNex
t:0' shape=(None,) dtype=string>, 'vehicle_fuel': <tf.Tensor 'IteratorGetNe
xt:1' shape=(None,) dtype=float32>, 'vehicle_noise': <tf.Tensor 'IteratorGe
tNext:2' shape=(None,) dtype=float32>, 'vehicle_speed': <tf.Tensor 'Iterato
rGetNext:3' shape=(None,) dtype=float32>, 'vehicle_type': <tf.Tensor 'Itera
torGetNext:4' shape=(None,) dtype=string>, 'vehicle_waiting': <tf.Tensor 'I
teratorGetNext:5' shape=(None,) dtype=float32>}. Consider rewriting this mo
del with the Functional API.
471/471 [==============================] - ETA: 0s - loss: 86672616.0000 -
mse: 86672616.0000WARNING:tensorflow:Layers in a Sequential model should on
ly have a single input tensor. Received: inputs={'vehicle_eclass': <tf.Tens
or 'IteratorGetNext:0' shape=(None,) dtype=string>, 'vehicle_fuel': <tf.Ten
sor 'IteratorGetNext:1' shape=(None,) dtype=float32>, 'vehicle_noise': <tf.
Tensor 'IteratorGetNext:2' shape=(None,) dtype=float32>, 'vehicle_speed':
<tf.Tensor 'IteratorGetNext:3' shape=(None,) dtype=float32>, 'vehicle_type
': <tf.Tensor 'IteratorGetNext:4' shape=(None,) dtype=string>, 'vehicle_wai
ting': <tf.Tensor 'IteratorGetNext:5' shape=(None,) dtype=float32>}. Consid
er rewriting this model with the Functional API.
471/471 [==============================] - 78s 163ms/step - loss: 86672616.
0000 - mse: 86672616.0000 - val_loss: 84066216.0000 - val_mse: 84066216.000
0
Epoch 2/90
471/471 [==============================] - 74s 158ms/step - loss: 72704192.
0000 - mse: 72704176.0000 - val_loss: 59197796.0000 - val_mse: 59197784.000
0
Epoch 3/90
471/471 [==============================] - 73s 156ms/step - loss: 54739364.
0000 - mse: 54739356.0000 - val_loss: 52062544.0000 - val_mse: 52062540.000
0
Epoch 4/90
471/471 [==============================] - 73s 155ms/step - loss: 47802824.
0000 - mse: 47802816.0000 - val_loss: 40719024.0000 - val_mse: 40719016.000
0
Epoch 5/90
471/471 [==============================] - 73s 155ms/step - loss: 25785734.
0000 - mse: 25785722.0000 - val_loss: 10064540.0000 - val_mse: 10064531.000
0
Epoch 6/90
471/471 [==============================] - 73s 156ms/step - loss: 4142423.0
000 - mse: 4142410.2500 - val_loss: 1774975.6250 - val_mse: 1774964.3750
Epoch 7/90
471/471 [==============================] - 74s 157ms/step - loss: 1245649.1
250 - mse: 1245639.0000 - val_loss: 875234.5000 - val_mse: 875223.3750
Epoch 8/90
471/471 [==============================] - 73s 156ms/step - loss: 644733.68
75 - mse: 644722.9375 - val_loss: 454780.0938 - val_mse: 454769.2188
Epoch 9/90
471/471 [==============================] - 73s 156ms/step - loss: 321679.21
```

```
88 - mse: 321668.1875 - val_loss: 212167.5000 - val_mse: 212156.4375
Epoch 10/90
471/471 [==============================] - 73s 155ms/step - loss: 138930.26
56 - mse: 138919.1875 - val_loss: 82189.6406 - val_mse: 82178.6172
Epoch 11/90
471/471 [==============================] - 73s 155ms/step - loss: 51121.503
9 - mse: 51110.4570 - val_loss: 28329.9629 - val_mse: 28318.9023
Epoch 12/90
471/471 [==============================] - 74s 158ms/step - loss: 17005.683
6 - mse: 16994.6426 - val_loss: 9046.0332 - val_mse: 9034.9805
Epoch 13/90
471/471 [==============================] - 72s 154ms/step - loss: 5480.0269
- mse: 5468.9702 - val_loss: 3165.7791 - val_mse: 3154.7207
Epoch 14/90
471/471 [==============================] - 73s 156ms/step - loss: 2138.4365
- mse: 2127.2881 - val_loss: 1446.4103 - val_mse: 1435.1986
Epoch 15/90
471/471 [==============================] - 74s 156ms/step - loss: 1162.8386
- mse: 1151.5989 - val_loss: 944.4260 - val_mse: 933.1191
Epoch 16/90
471/471 [==============================] - 73s 156ms/step - loss: 793.6353
- mse: 782.2634 - val_loss: 678.3741 - val_mse: 666.9532
Epoch 17/90
471/471 [==============================] - 72s 154ms/step - loss: 605.0005
- mse: 593.5502 - val_loss: 552.3653 - val_mse: 540.8934
Epoch 18/90
471/471 [==============================] - 73s 156ms/step - loss: 510.6439
- mse: 499.1603 - val_loss: 480.5630 - val_mse: 469.0717
Epoch 19/90
471/471 [==============================] - 73s 156ms/step - loss: 455.6129
- mse: 444.1219 - val_loss: 438.5973 - val_mse: 427.1085
Epoch 20/90
471/471 [==============================] - 73s 155ms/step - loss: 417.6887
- mse: 406.2007 - val_loss: 403.4166 - val_mse: 391.9304
Epoch 21/90
471/471 [==============================] - 73s 155ms/step - loss: 381.0652
- mse: 369.5825 - val_loss: 362.1320 - val_mse: 350.6529
Epoch 22/90
471/471 [==============================] - 73s 155ms/step - loss: 346.6116
- mse: 335.1376 - val_loss: 331.6906 - val_mse: 320.2218
Epoch 23/90
471/471 [==============================] - 73s 156ms/step - loss: 321.6197
- mse: 310.1569 - val_loss: 308.1672 - val_mse: 296.7108
Epoch 24/90
471/471 [==============================] - 73s 155ms/step - loss: 298.5596
- mse: 287.1101 - val_loss: 285.1111 - val_mse: 273.6684
Epoch 25/90
471/471 [==============================] - 73s 156ms/step - loss: 276.3342
- mse: 264.8989 - val_loss: 272.0602 - val_mse: 260.6329
Epoch 26/90
471/471 [==============================] - 73s 156ms/step - loss: 253.6262
- mse: 242.2071 - val_loss: 239.1563 - val_mse: 227.7462
Epoch 27/90
471/471 [==============================] - 73s 156ms/step - loss: 231.0986
- mse: 219.6982 - val_loss: 218.1855 - val_mse: 206.7953
Epoch 28/90
471/471 [==============================] - 73s 156ms/step - loss: 210.1543
- mse: 198.7730 - val_loss: 207.1952 - val_mse: 195.8249
Epoch 29/90
```

```
471/471 [==============================] - 72s 154ms/step - loss: 191.0463
- mse: 179.6877 - val_loss: 179.1763 - val_mse: 167.8295
Epoch 30/90
471/471 [==============================] - 73s 156ms/step - loss: 174.2289
- mse: 162.8929 - val_loss: 180.8000 - val_mse: 169.4738
Epoch 31/90
471/471 [==============================] - 73s 155ms/step - loss: 159.4108
- mse: 148.0939 - val_loss: 158.8944 - val_mse: 147.5870
Epoch 32/90
471/471 [==============================] - 73s 156ms/step - loss: 145.2255
- mse: 133.9261 - val_loss: 147.3602 - val_mse: 136.0683
Epoch 33/90
471/471 [==============================] - 72s 154ms/step - loss: 135.9965
- mse: 124.7102 - val_loss: 147.8809 - val_mse: 136.5980
Epoch 34/90
471/471 [==============================] - 73s 156ms/step - loss: 130.0001
- mse: 118.7159 - val_loss: 123.2183 - val_mse: 111.9331
Epoch 35/90
471/471 [==============================] - 73s 155ms/step - loss: 124.1202
- mse: 112.8356 - val_loss: 117.8675 - val_mse: 106.5842
Epoch 36/90
471/471 [==============================] - 74s 156ms/step - loss: 119.2305
- mse: 107.9472 - val_loss: 119.6500 - val_mse: 108.3672
Epoch 37/90
471/471 [==============================] - 73s 155ms/step - loss: 116.7077
- mse: 105.4262 - val_loss: 111.2964 - val_mse: 100.0167
Epoch 38/90
471/471 [==============================] - 73s 156ms/step - loss: 112.2682
- mse: 100.9911 - val_loss: 110.9699 - val_mse: 99.6970
Epoch 39/90
471/471 [==============================] - 73s 156ms/step - loss: 110.6239
- mse: 99.3557 - val_loss: 109.0032 - val_mse: 97.7399
Epoch 40/90
471/471 [==============================] - 73s 155ms/step - loss: 108.2708
- mse: 97.0142 - val_loss: 102.4028 - val_mse: 91.1532
Epoch 41/90
471/471 [==============================] - 73s 156ms/step - loss: 105.9380
- mse: 94.6951 - val_loss: 105.7912 - val_mse: 94.5556
Epoch 42/90
471/471 [==============================] - 75s 159ms/step - loss: 103.9877
- mse: 92.7596 - val_loss: 107.1695 - val_mse: 95.9486
Epoch 43/90
471/471 [==============================] - 72s 154ms/step - loss: 100.8782
- mse: 89.6657 - val_loss: 97.4469 - val_mse: 86.2429
Epoch 44/90
471/471 [==============================] - 73s 155ms/step - loss: 98.4496 -
mse: 87.2542 - val_loss: 106.6973 - val_mse: 95.5111
Epoch 45/90
471/471 [==============================] - 73s 155ms/step - loss: 95.6886 -
mse: 84.5101 - val_loss: 127.1995 - val_mse: 116.0299
Epoch 46/90
471/471 [==============================] - 73s 155ms/step - loss: 93.7635 -
mse: 82.6032 - val_loss: 107.1678 - val_mse: 96.0161
Epoch 47/90
471/471 [==============================] - 73s 154ms/step - loss: 90.9115 -
mse: 79.7722 - val_loss: 95.0634 - val_mse: 83.9353
Epoch 48/90
471/471 [==============================] - 73s 156ms/step - loss: 90.5228 -
mse: 79.4010 - val_loss: 94.1196 - val_mse: 82.9999
```

```
Epoch 49/90
471/471 [==============================] - 73s 156ms/step - loss: 89.0788 -
mse: 77.9632 - val_loss: 84.3685 - val_mse: 73.2581
Epoch 50/90
471/471 [==============================] - 73s 155ms/step - loss: 87.7064 -
mse: 76.6039 - val_loss: 85.7556 - val_mse: 74.6606
Epoch 51/90
471/471 [==============================] - 74s 157ms/step - loss: 87.0640 -
mse: 75.9739 - val_loss: 88.3806 - val_mse: 77.2864
Epoch 52/90
471/471 [==============================] - 73s 155ms/step - loss: 86.2873 -
mse: 75.1779 - val_loss: 81.4625 - val_mse: 70.3361
Epoch 53/90
471/471 [==============================] - 74s 157ms/step - loss: 85.5224 -
mse: 74.3843 - val_loss: 80.7414 - val_mse: 69.5968
Epoch 54/90
471/471 [==============================] - 72s 153ms/step - loss: 83.9426 -
mse: 72.7965 - val_loss: 82.8244 - val_mse: 71.6788
Epoch 55/90
471/471 [==============================] - 74s 156ms/step - loss: 83.5796 -
mse: 72.4347 - val_loss: 80.5091 - val_mse: 69.3662
Epoch 56/90
471/471 [==============================] - 73s 155ms/step - loss: 82.7181 -
mse: 71.5773 - val_loss: 79.4703 - val_mse: 68.3314
Epoch 57/90
471/471 [==============================] - 73s 156ms/step - loss: 82.2991 -
mse: 71.1624 - val_loss: 80.9644 - val_mse: 69.8297
Epoch 58/90
471/471 [==============================] - 73s 155ms/step - loss: 81.8291 -
mse: 70.6960 - val_loss: 79.1921 - val_mse: 68.0605
Epoch 59/90
471/471 [==============================] - 73s 155ms/step - loss: 81.5122 -
mse: 70.3818 - val_loss: 85.1062 - val_mse: 73.9770
Epoch 60/90
471/471 [==============================] - 73s 156ms/step - loss: 80.2711 -
mse: 69.1437 - val_loss: 79.0197 - val_mse: 67.8934
Epoch 61/90
471/471 [==============================] - 73s 156ms/step - loss: 80.7846 -
mse: 69.6597 - val_loss: 78.4975 - val_mse: 67.3747
Epoch 62/90
471/471 [==============================] - 73s 155ms/step - loss: 80.2030 -
mse: 69.0814 - val_loss: 81.6328 - val_mse: 70.5134
Epoch 63/90
471/471 [==============================] - 73s 154ms/step - loss: 79.0899 -
mse: 67.9722 - val_loss: 76.9707 - val_mse: 65.8548
Epoch 64/90
471/471 [==============================] - 72s 152ms/step - loss: 78.9234 -
mse: 67.8088 - val_loss: 75.4734 - val_mse: 64.3596
Epoch 65/90
471/471 [==============================] - 74s 156ms/step - loss: 78.1448 -
mse: 67.0311 - val_loss: 81.5893 - val_mse: 70.4760
Epoch 66/90
471/471 [==============================] - 74s 156ms/step - loss: 78.2026 -
mse: 67.0903 - val_loss: 73.3045 - val_mse: 62.1925
Epoch 67/90
471/471 [==============================] - 73s 156ms/step - loss: 76.6422 -
mse: 65.5309 - val_loss: 72.9820 - val_mse: 61.8718
Epoch 68/90
471/471 [==============================] - 74s 157ms/step - loss: 76.1423 -
```

```
mse: 65.0324 - val_loss: 74.9498 - val_mse: 63.8401
Epoch 69/90
471/471 [==============================] - 73s 155ms/step - loss: 76.2369 -
mse: 65.1276 - val_loss: 71.7436 - val_mse: 60.6346
Epoch 70/90
471/471 [==============================] - 73s 155ms/step - loss: 75.6724 -
mse: 64.5638 - val_loss: 71.6086 - val_mse: 60.4999
Epoch 71/90
471/471 [==============================] - 73s 155ms/step - loss: 74.7880 -
mse: 63.6796 - val_loss: 75.2440 - val_mse: 64.1355
Epoch 72/90
471/471 [==============================] - 74s 157ms/step - loss: 74.2735 -
mse: 63.1646 - val_loss: 78.0767 - val_mse: 66.9664
Epoch 73/90
471/471 [==============================] - 73s 156ms/step - loss: 73.1167 -
mse: 62.0060 - val_loss: 79.2722 - val_mse: 68.1613
Epoch 74/90
471/471 [==============================] - 73s 156ms/step - loss: 73.0882 -
mse: 61.9762 - val_loss: 68.8879 - val_mse: 57.7755
Epoch 75/90
471/471 [==============================] - 73s 155ms/step - loss: 72.2048 -
mse: 61.0917 - val_loss: 68.4172 - val_mse: 57.3035
Epoch 76/90
471/471 [==============================] - 73s 156ms/step - loss: 72.1961 -
mse: 61.0819 - val_loss: 67.6088 - val_mse: 56.4939
Epoch 77/90
471/471 [==============================] - 74s 157ms/step - loss: 71.1881 -
mse: 60.0723 - val_loss: 67.6177 - val_mse: 56.5008
Epoch 78/90
471/471 [==============================] - 74s 158ms/step - loss: 70.8034 -
mse: 59.6859 - val_loss: 68.8632 - val_mse: 57.7445
Epoch 79/90
471/471 [==============================] - 73s 155ms/step - loss: 70.3519 -
mse: 59.2324 - val_loss: 75.4520 - val_mse: 64.3314
Epoch 80/90
471/471 [==============================] - 73s 156ms/step - loss: 69.2807 -
mse: 58.1596 - val_loss: 66.0964 - val_mse: 54.9747
Epoch 81/90
471/471 [==============================] - 73s 156ms/step - loss: 68.8834 -
mse: 57.7604 - val_loss: 68.2197 - val_mse: 57.0956
Epoch 82/90
471/471 [==============================] - 73s 156ms/step - loss: 69.1446 -
mse: 58.0192 - val_loss: 65.2923 - val_mse: 54.1652
Epoch 83/90
471/471 [==============================] - 72s 154ms/step - loss: 67.6454 -
mse: 56.5176 - val_loss: 82.0331 - val_mse: 70.9041
Epoch 84/90
471/471 [==============================] - 73s 156ms/step - loss: 67.0947 -
mse: 55.9646 - val_loss: 71.9856 - val_mse: 60.8547
Epoch 85/90
471/471 [==============================] - 73s 155ms/step - loss: 67.2501 -
mse: 56.1183 - val_loss: 62.8737 - val_mse: 51.7408
Epoch 86/90
471/471 [==============================] - 73s 155ms/step - loss: 66.3339 -
mse: 55.1999 - val_loss: 71.5279 - val_mse: 60.3936
Epoch 87/90
471/471 [==============================] - 73s 154ms/step - loss: 66.0985 -
mse: 54.9626 - val_loss: 71.5925 - val_mse: 60.4558
Epoch 88/90
```

```
471/471 [==============================] - 74s 156ms/step - loss: 65.5822 -
mse: 54.4444 - val_loss: 71.4693 - val_mse: 60.3300
Epoch 89/90
471/471 [==============================] - 73s 154ms/step - loss: 65.1314 -
mse: 53.9916 - val_loss: 64.3657 - val_mse: 53.2241
Epoch 90/90
471/471 [==============================] - 74s 156ms/step - loss: 63.7811 -
mse: 52.6383 - val_loss: 65.2633 - val_mse: 54.1198
```

Out[8]:
```
<keras.callbacks.History at 0x7f777d6eefa0>
```

Evaluate the Model with Test Data

As a final step, I have implemented several charts to view the models ability to predict CO2 vehicle emissions.

In [9]:
```
test_lbl = np.array(test_df["vehicle_CO2"])
test_df = test_df.drop(columns=["vehicle_CO2"])
test_ft = {key:np.array(value) for key, value in test_df.items()}
print("Model evaluation: \n")
model.evaluate(x=test_ft, y=test_lbl, batch_size=batch_size)
```

```
Model evaluation:

66/66 [==============================] - 2s 27ms/step - loss: 64.9543 - ms
e: 53.8107
```

Out[9]:
```
[64.95425415039062, 53.81074523925781]
```

In [10]:
```
# Get a summary of the model
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 Features (DenseFeatures)    multiple                  0

 Hidden1 (Dense)             multiple                  336

 Hidden2 (Dense)             multiple                  400

 Hidden3 (Dense)             multiple                  187

 Hidden4 (Dense)             multiple                  60

 Output (Dense)              multiple                  6

=================================================================
Total params: 989
Trainable params: 989
Non-trainable params: 0
_________________________________________________________________
```

In [11]:
```
%tensorboard --logdir logs/fit
```

TensorBoard UPLOAD

Data could not be loaded.

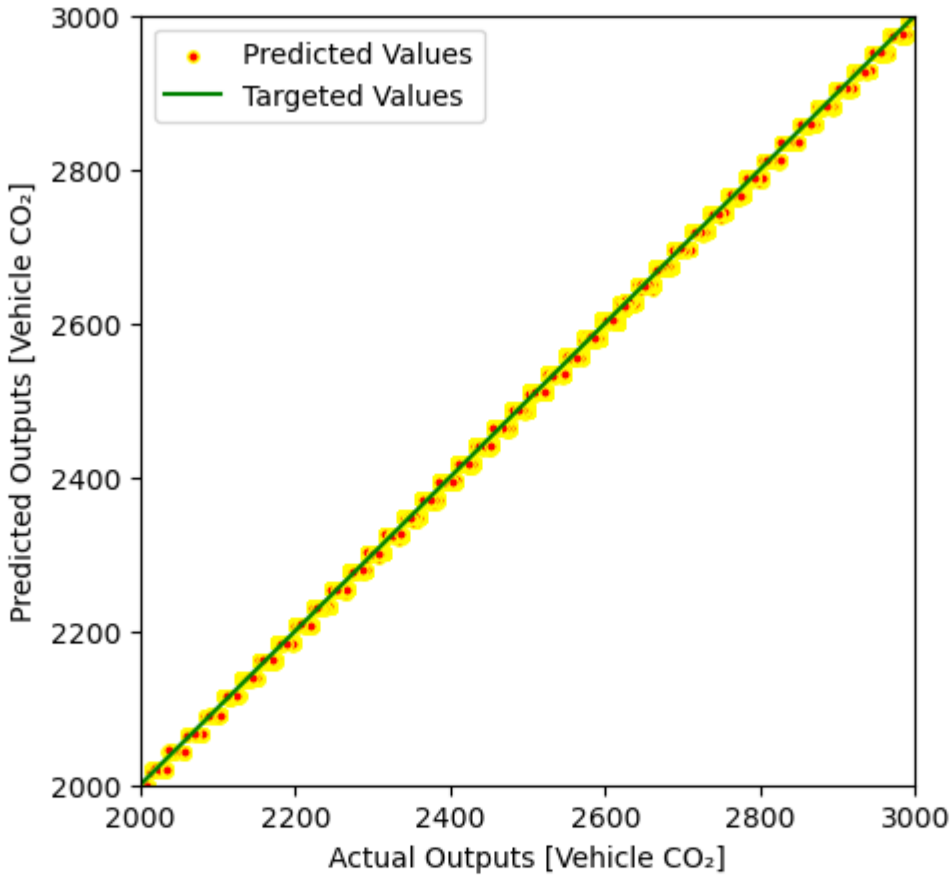
The TensorBoard server may be down or inaccessible.

Last reload:

In [12]:
```
%%time
# Get the features from the test set
test_features = test_ft
# Get the actual CO2 output for the test set
actual_labels = test_lbl

# Make prediction on the test set
predicted_labels = model.predict(x=test_features).flatten()

# Define the graph
Figure1 = plt.figure(figsize=(5,5), dpi=100)
plt.xlabel('Actual Outputs [Vehicle CO\u2082]')
plt.ylabel('Predicted Outputs [Vehicle CO\u2082]')
plt.scatter(actual_labels, predicted_labels, s=15, c='Red', edgecolors='Yell

# Take the output data from 2000 to 3000 as an instance to visualize
lims = [2000, 3000]
plt.xlim(lims)
plt.ylim(lims)
plt.plot(lims, lims, color='Green', label='Targeted Values')
plt.legend()
```

```
WARNING:tensorflow:Layers in a Sequential model should only have a single i
nput tensor. Received: inputs={'vehicle_eclass': <tf.Tensor 'IteratorGetNex
t:0' shape=(None,) dtype=string>, 'vehicle_fuel': <tf.Tensor 'IteratorGetNe
xt:1' shape=(None,) dtype=float32>, 'vehicle_noise': <tf.Tensor 'IteratorGe
tNext:2' shape=(None,) dtype=float32>, 'vehicle_speed': <tf.Tensor 'Iterato
rGetNext:3' shape=(None,) dtype=float32>, 'vehicle_type': <tf.Tensor 'Itera
torGetNext:4' shape=(None,) dtype=string>, 'vehicle_waiting': <tf.Tensor 'I
teratorGetNext:5' shape=(None,) dtype=float32>}. Consider rewriting this mo
del with the Functional API.
51035/51035 [==============================] - 1317s 26ms/step
CPU times: user 1h 30min 25s, sys: 1h 16min 54s, total: 2h 47min 20s
Wall time: 22min 12s
```

Out[12]:
```
<matplotlib.legend.Legend at 0x7f777c5fc820>
```



Error Count Histogram

In [13]:
```
error = actual_labels - predicted_labels
Figure2 = plt.figure(figsize=(8,3), dpi=100)
plt.hist(error, bins=50, color='Red', edgecolor='Green')
plt.xlabel('Prediction Error [Vehicle CO\u2082]')
plt.ylabel('Count')
```

Out[13]:
```
Text(0, 0.5, 'Count')
```

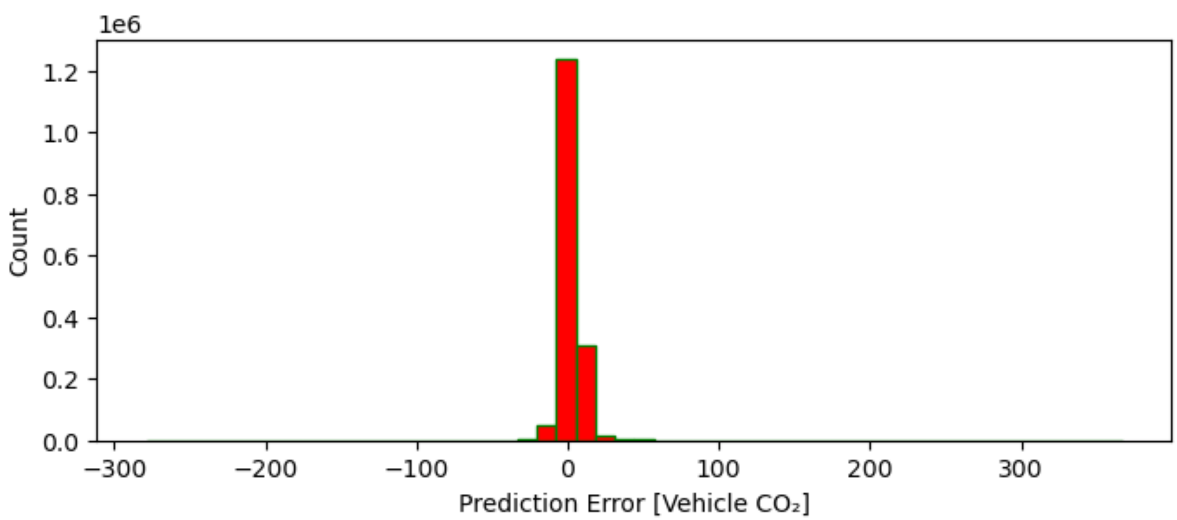


Table of Actual and Predicted Values

Below, a table puts the actual and predicted values side by side. Html is used in this case.

In [14]:
```
from IPython.display import HTML, display

def display_table(data_x, data_y):
    html = "<table>"
    html += "<tr>"
    html += "<td><h3>%s</h3><td>"%"Actual Vehicle CO\u2082"
    html += "<td><h3>%s</h3><td>"%"Predicted Vehicle CO\u2082"
    html += "</tr>"
    for i in range(len(data_x)):
        html += "<tr>"
        html += "<td><h4>%s</h4><td>"%(int(data_x[i]))
        html += "<td><h4>%s</h4><td>"%(int(data_y[i]))
        html += "</tr>"
    html += "</table>"
    display(HTML(html))

display_table(actual_labels[0:100], predicted_labels[0:100])
```

Actual Vehicle CO_2	Predicted Vehicle CO_2
5286	5279
6607	6603
2624	2620
10793	10791
5296	5301
11062	11071
5867	5859
9746	9744
2707	2696
5689	5694
6935	6927
0	0
2839	2836
0	0
20898	20915
0	0
0	0
2698	2697
7610	7603
9341	9348
2624	2622
2624	2622

4729	4716
0	0
3854	3858
2624	2622
0	0
17305	17313
2624	2622
0	0
6130	6138
7319	7322
2667	2672
0	0
0	0
49704	49687
2624	2621
8525	8511
0	0
0	0
5855	5857
2453	2441
7485	7486
0	0
4222	4230

5632	5625
0	0
4592	4580
8762	8767
0	0
6140	6138
6554	6556
0	0
0	0
0	0
4707	4696
2624	2620
0	0
2356	2349
0	0
5286	5283
9492	9488
10527	10535
0	0
6575	6580
0	0
0	0
6717	6720

2547	2556
0	0
2886	2882
0	0
0	0
6895	6882
0	0
11575	11583
5286	5289
4051	4044
7784	7789
42675	42660
0	0
0	0
0	0
0	0
13810	13815
0	0
0	0
10477	10464
3674	3673
67610	67584
6527	6533

0	0
0	0
0	0
4545	4533
0	0
0	0
10194	10186
6265	6254
5286	5289